The Virtual Path: The Domain Model for the Design of the MIRO Surgical Robotic System

Mathias Nickl * **Stefan Jörg** * **Gerd Hirzinger** *

* *German Aerospace Center (DLR e.V.), Inst. of Robotics and Mechatronics, Germany (e-mail: mathias.nickl@dlr.de)*

Abstract:
The MIRO Platform, designed at DLR, is a highly integrated and compliant mechatronic robotic system for minimally invasive surgery. This publication presents the *"Virtual Path,"* a domain model in the sense of Domain Driven Design, which provides a formal guideline for all designers of the MIRO Platform to obtain a deterministic implementation without the necessity of a monolithic framework. Using Hardware-Software Co-Design methods, the roles of the basic component types of a robotic system were formally defined. The result is a set of design idioms that provide a solution for three main issues when mapping those components to a distributed heterogeneous mechatronic architecture: synchronization, scheduling, and error handling.

Keywords: Interdisciplinary Design, Surgical Robots, Failure Detection, Synchronization

1. INTRODUCTION

Minimally invasive surgery (MIS) reduces pain and trauma, loss of blood, and the risk of wound infections, since it involves incisions that are considerably smaller than those in open surgery. Robotics is expected to advance the predictability and accuracy of MIS. However, surgery is a complex scenario, where surgeons and assistants are well-rehearsed, and space is insufficient in already crowded operation rooms. This asks for highly integrated and compliant robotic systems.

Considering this, DLR's Institute of Robotics and Mechatronics has developed a robotic surgery system: the *MIRO Platform* (see Fig. 1). The core of the system consists of three MIRO robot arms that are directly mounted on the operation table. Depending on the medical procedure, this core platform can be extended with dedicated instruments, such as endoscopes or actuated tongs.

The *MIRO* system is a research platform, i.e. surgical procedures as well as the robotic platform are involved in a continuous design process where the mechatronic components and the system's configuration is continuously changed by the designers. Nevertheless, unpredictable behavior due to these modifications is not acceptable, since determinism is essential for medical surgery applications. For deterministic behavior, robust control and a reliable hardware platform are as important as a deterministic implementation process.

Problem

A look at the design practice at our institute reveals that *algorithmic design* and *infrastructure design* are different tasks. Designers with dedicated expertise in control design create an *algorithmic model*, which is an executable model

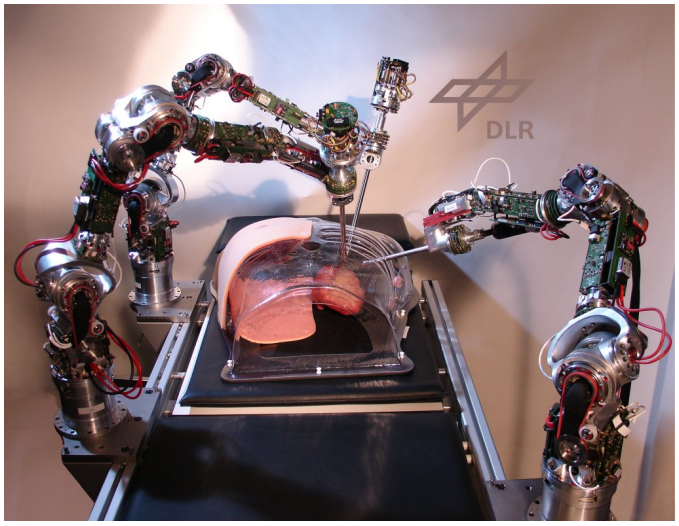

Fig. 1. The MIRO Platform for minimally invasive surgery (Hagn et al. (2008))

that represents the control laws. Then they use code-generators such as Mathwork's Real-time Workshop and HDL-Coder to map the *algorithmic model* to dedicated *target technologies*, e.g. CPUs, FPGAs, etc. However, a deterministic implementation result is not simply a composition of these artifacts, since the *algorithmic model* deliberately ignores important implementation details such as synchronization to physical time, causal scheduling in face of distribution, and global error handling to ensure determinism in case of exceptions. Hence, there is a gap between the algorithmic specification and a reliable mechatronic system. This gap has to be filled by the system's *infrastructure design*.

To reach a high level of integration, the mechanic and electronic components of the MIRO arms are customly designed in-house. Augmented by standard communication and computing hosts, the MIRO system is a heterogeneous platform that combines small footprint sensor and actuator devices with powerful computing resources. This approach yields the flexibility that is required for a research platform. The result is a mechatronic system with *Field-Programmable Gate Arrays* (FPGAs) for communication and motor control, *Complex Programmable Logic Devices* (CPLDs) for high integrated sensor implementations, and *Intel Pentium CPUs* for robot control and operation planning.

Hence, the MIRO-system is an open, heterogeneous, distributed system with strict real-time constraints. An infrastructure framework for such a system has to cover the system's complexity and the changing requirements of research projects. We do not intend to build a monolithic framework, since framework design is always unwieldy. Moreover, generic approaches tend to over-specifiy without regarding the dedicated domain-knowledge. In contrast, the authors believe that only light-weight concepts are flexible enough for an agile system design process.

Solution

Domain Driven Design, an instance of model-driven design, introduced by Evans (2004), is an approach for tackling complexity by a platform-independent *domain model*. The *domain model* captures the underlying structure of an implementation domain, concentrates the domain-specific knowledge, and acts as a *ubiquitous language* for all system designers.

The intent of this paper is to present the *domain model*, *"Virtual Path,"* that has been developed to handle the complexity of the mechatronic components of DLR's MIRO system. The *domain model* serves as an implementation guideline for the design of controllers, mechatronic components, and dedicated tools for a deterministic refinement process. Therefore, the *"Virtual Path"* provides the following formal specifications for modeling and implementation:

- The *synchronous model* is the underlying *model of computation*, which is the specification and simulation platform for control designers.
- The *refinement process* of the MIRO-system maps algorithmic models to heterogeneous *target platforms*. The *algorithmic models*, and the description of the target architecture, called *architecture models*, define the *design space*, i.e. the set of all valid implementations, which serve as a formal frame for an iterative optimization process (see Fig. 2).
- The *model of interaction* specifies the *infrastructure* details that have to be considered by all designers to guarantee determinism for interacting mechatronic components. More precisely, it specifies synchronization, scheduling and global error handling.

The *Virtual Path* identifies the domain-specific knowledge for mechatronic systems and provides a set of definitions, which serve as a frame for a reliable implementation process. The result is an open workbench with suitable

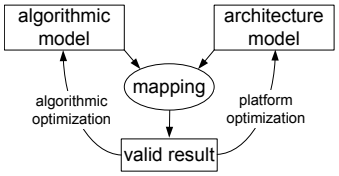

Fig. 2. The implementation process of the *Virtual Path*

commercial and custom tools for the various implementation tasks. This presentation focuses on the *model of interaction*, since the *synchronous model* is discussed in detail by the synchronous language community (see below) and the *refinement process* of the *Virtual Path* will be published separately.

State of the Art

Kienhuis et al. (2001) have introduced a scheme called *"Y-Chart"* for the concerted optimization of hardware and software components by an iterative mapping process. Their ideas have strong influence on the presented *domain model*. However, the *Y-Chart* focuses on a wide scope of applications where automatic optimization is used to cope with a huge design-space. In contrast, the *Virtual Path* reduces the design-space by using domain knowledge.

A common solution for *infrastructure design* is a *safe platform approach*, which assumes that determinism is given by the *target platform*. This means that the *target platform* provides global synchronization and error handling mechanisms. For example, Real-Time-OS (e.g. QNX or VxWorks) and deterministic field-buses (e.g. Time Triggered Protocol [Kopetz and Bauer (2001)]) lead to a homogeneous *target platform* with consistent system-wide protocols. Hence, an *algorithmic model* can be mapped with little effort to these platforms. However, the *safe platform approach* is too rigid for the design of a complex system such as the MIRO platform. Especially when a high level of integration is required, a platform could not be realized with only unified system-wide protocols and standard communication interfaces.

It is common sense to describe digital control systems with a *synchronous model*, i.e. with discrete signals that assume a fixed sample period, T_s. Commonly, the computation time of control algorithms is ignored, since the computation path is considered to be calculated in zero time [Franklin et al. (1998)]. This idealized view of instantaneous system reaction is known as the "Synchronous Hypothesis" and has been described formally by the synchronous language community [Benveniste and Berry (2001)]. Several projects support control design based on the *synchronous model*. *ESTREL* provides the development framework *SCADE*, which includes a formal code generation process for *homogeneous platforms*. Mathwork's *Simulink* supports synchronous models and provides the Real-Time Workshop to be used for generating code for certain platforms. Neither framework supports distributed target platforms or communication refinement.

The Ptolemy Project [Eker et al. (2003)] realizes a universal framework for embedded system design that focuses on heterogeneous modeling and simulation. Ptolemy is suitable for signal oriented control design, but the support for code generation is minimal.

Metropolis [Balarin et al. (2003)] is a HW/SW co-design framework which focuses on abstract communication modeling and deterministic implementation refinement. A strong time-triggered concept enables the precise specification of calculation and communication delays. However, the demand for accurate timing models that specify these delays is too rigid for an agile prototyping design process.

In summary, many approaches try to find common solutions for modeling and code refinement for embedded systems. However, none of them addresses all of the following requirements: openness, heterogeneity, distribution, real-time, and an agile design process. A *domain model* for the design of mechatronic systems is a valuable alternative to a monolithic all-in-one framework. Section 2 describes the underlying concept of the *Virtual Path*. Sections 3 to 5 discuss the main aspects of the *model of interaction*: synchronization, scheduling and error handling.

2. THE DOMAIN MODEL "VIRTUAL PATH"

The notion of the *Virtual Path* is derived from the discrete part of a feedback control loop, where the control law is implemented. We regard the *computation path* in the discrete world as a virtual path that starts at a *Sensor*, passes through a *Controller*, and terminates at an *Actuator* (see Fig. 3). Thus, the *"Virtual Path"* starts in the physical world and goes back to the physical world. The boundaries between the discrete, virtual world and the continuous, physical world are the A/D converters of the sensor signals and the motor electronics driving an actuator (see Fig 4). Hence, the *Virtual Path* defines a unidirectional discrete signal flow.

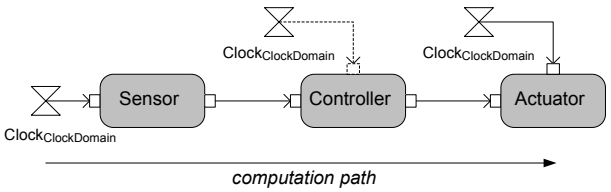

Fig. 3. The *Virtual Path* starts and terminates in the physical world.

Typical mechatronic systems consist of multiple *Sensors*, *Controllers*, and *Actuators*, such as the MIRO coupled wrist joint, which consists of position and torque sensors, a cascade of impedance and motor current controllers, and two actuators (see Fig. 5). In fact, a *Virtual Path* is not a single path from *Sensor* to *Actuator* but rather a complex net of parallel and sequential *computation paths*.

Fig. 3 depicts the most simple *Virtual Path* that consists of only one *computation path*. A *computation path* denotes the order of causal calculations of concurrent components. It begins at a global clock and goes with the signal flow. Hence, a net of parallel and sequential *computation paths* determines the scheduling scheme of a *Virtual Path*.

The *Virtual Path* is based on *actor-oriented design*, a component based *model of concurrency* for signal oriented systems [Lee et al. (2003)]. Here, components interact only by messages, called *firings*, where incoming *firings* trigger the execution of a component's *behavior*. Formally, a *behavior* is defined as a function that is executed by

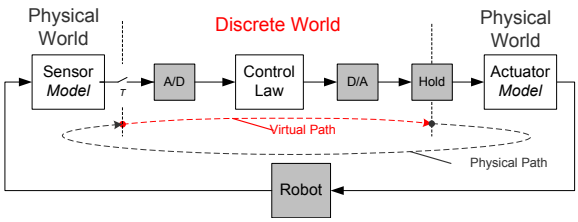

Fig. 4. The *Virtual Path* is the control loop's part that is implemented on a discrete platform.

a *process* [Agha (1986)]. Thus, actor-oriented design synchronizes concurrent processes merely by messages.

The *Virtual Path* specifies *Sensor*, *Controller*, *Actuator*, and *Signal* as domain-specific roles that provide an implementation guideline for both software and hardware designers of mechatronic components. *Sensors* sample the physical world and synchronize the implementation to *physical time* (see Sec. 3). *Controllers* implement control algorithms and guarantee causality, i.e. the order of distributed calculations (see Sec. 4). *Actuators* terminate a *Virtual Path* and constitute the interface that impacts the *physical world*. To guarantee determinism, *Actuators* observe the results of a *Virtual Path* and implement an efficient error handling (see Sec. 5). Finally, *Signals* connect components and are represented by the transmission of *events*.

The *Virtual Path* is *event-driven*. According to the control designers' view, a signal within a *Virtual Path* is a sequence of discrete events where an event is defined as the value at a discrete point of time. A formal definition is introduced by Lee and Sangiovanni-Vincentelli (1998), who describe an event as the tuple of a tag and a value. In the context of the *Virtual Path*, the tag is a *timestamp*, $t[n]$, and the *value*, $v[n]$, is augmented by a *quality*, $q[n]$, which is laid out in Sec. 5. Thus, an event, $e[n]$, is defined as the following:

$$e[n] = (t[n], v[n], q[n]), n \in \mathbb{N}_0. \quad (1)$$

A signal s can be viewed as a subset of $T \times V \times Q$, where T is the set of *timestamps*, V is the set of *values*, and Q is the set of *qualities*. V and Q include a special symbol $\perp$ (called "bottom"), to denote invalid calculation results caused by runtime errors, e.g. $e[n] = (t[n], \perp, q[n])$ reflects an event with invalid *value* and valid *quality*.

The *synchronous model* defines the *Virtual Path*'s underlying *model of computation*, which is the specification

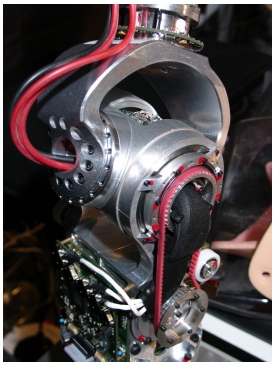

Fig. 5. MIRO's shoulder, a coupled joint with 2 DoF

and simulation platform for control designers [Lee and Sangiovanni-Vincentelli (1998)]. It predetermines domain-specific implementation constraints, which essentially reduce the complexity of an implementation. In particular, the clear scheduling scheme of the *synchronous model*, which constitutes exactly one single execution of the *computation path* per clock cycle, considerably simplifies its implementation.

The *Virtual Path* guarantees determinism in terms of real-time. The *synchronous model* assumes calculation of the *computation path* in zero time, whereas computations on a real system introduce delay. This deviation from the synchronous hypothesis affects the stability of feedback control loops and has to be considered to guarantee determinism. To accomplish this, the execution time of the *computation path* is measured continuously.

The *Virtual Path* provides global error handling for distributed components without requiring a monolithic framework. Errors that occur at one location generally affect other parts of the system as well. Hence, the *Virtual Path* defines an effective error propagation scheme based on the natural signal flow from *Sensors* to *Actuators* and an error handling scheme for *actuation units*, i.e. the coordination of actuators that drive coupled mechanic components (see Sec. 5).

The following sections discuss the explicit roles of *Sensors*, *Controllers*, and *Actuators* and illustrate that global synchronization, local scheduling, and error handling are the main pillars for a *domain model* that specifies the interaction of mechatronic components.

3. GLOBAL SYNCHRONIZATION

It is important to distinguish two very close notions of synchronization, since the implementations of the appropriate synchronization mechanisms differ essentially. The parallel computing community regards synchronization as a blocking mechanism that handles race conditions between concurrent processes [Hoare (1978)]. For mechatronic systems, a more natural notion is important as well: Synchronization in the sense of "at the same time," where time is the physical time, $t_{physical}$, of a system. Synchronization in this sense demands the implementation of a *physical clock* which is a discrete representation of *physical time*. In accordance with Lamport (1978), who introduced a formal definition of a *physical clock* as ordered sets, we use the term *synchronization* to mean the latter sense.

The *physical clock* is the common base for all algorithmic calculations in a synchronous real-time system. For this

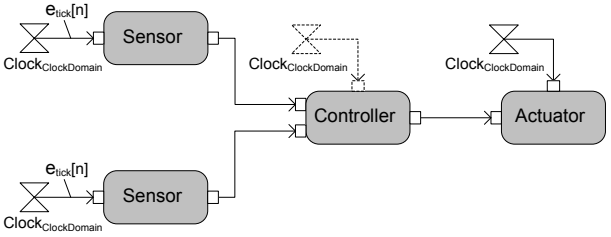

Fig. 6. The *Virtual Path* must be synchronized with physical time.

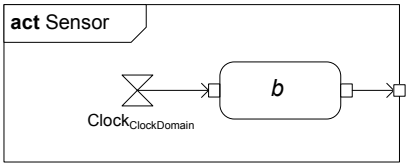

Fig. 7. A *Sensor* implementation requires a clock.

reason, calculations are triggered by periodic synchronization events, or *ticks*, which are synchronous with $t_{physical}$. A *tick* is a tuple containing only a timestamp:

$$e_{tick}[n] = t[n], n \in \mathbb{N}_0, \quad (2)$$

The relation of $t[n]$ to $t_{Physical}$ is defined by a *Clock Domain*, where T_s is the *sample period* and θ is the phase offset in relation to $t_{Physical}$:

$$t[n] = (n \cdot T_s) + \theta, n \in \mathbb{N}_0 \quad (3)$$

A *computation path* resides in a certain *Clock Domain*, i.e. all clocks have the same T_s and θ. Ideally, all *computation paths* of a *Virtual Path* have the same *Clock Domain*. Otherwise, multiple *Clock Domains* have to be connected with dedicated rate transitions, which are filters that reside in both neighboring *Clock Domains*. These filters decouple the *computation paths* of different *Clock Domains*.

Implementations of a *physical clock* are numerous, since synchronization and representation of time are highly platform-dependent. Hence, the *Virtual Path* introduces an abstract concept of a *Global Clock* that hides all platform-dependent details. The *Global Clock* is considered to be a part of the communication infrastructure, which provides the synchronization events, e_{tick}, that trigger the *computation path*. A *tick-event* is represented as a pulse, a unidirectional non-blocking message that contains only a timestamp.

With the known *Clock Domain* of Eqn. 3 the platform specific representation of a *timestamp*, $t[n]$, is n, a simple positive integer counter.

Fig. 7 depicts the structure of a *Sensor*, where a *Global Clock* triggers the execution of the *Sensor*'s behavior, b, that implements the sensor's measurement.

4. LOCAL SCHEDULING

As mentioned before, it is important to distinguish two notions of synchronization. One is synchronization in terms of time (see section 3). The other regards the coordination of parallel processes to enable the causal distribution of calculations. Synchronization in the latter context demands the implementation of inter-process-communication and not necessarily a concept of time. The embedded systems community uses the term *scheduling* for this notion, and various *models of concurrency* define this concept formally. Hence, *scheduling* is not a mechanism to share single processors, but a scheme to organize the execution order of concurrent processes [Lee and Sangiovanni-Vincentelli (1998)].

Local scheduling is equivalent to the aggregation of received events. Aggregation of events means generating one single event with timestamp and aggregated values of all events that are related to the same timestamp. Therefore,

$$e_{aggr}[n] = (t[n], (v_0[n], .., v_{i-1}[n]), (q_0[n], .., q_{i-1}[n])) \quad (4)$$

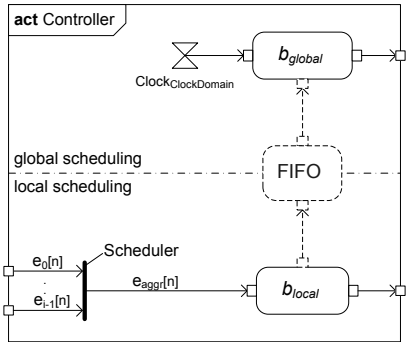

Fig. 8. A *Controller* can be synchronized globally by *ticks* or locally by input events.

where $i \in \mathbb{N}_0$ is the known number of input signals. Thus, local scheduling joins multiple *calculation paths*.

A *Controller's* behavior can be synchronized globally by *ticks* or locally by input events. This is reflected by the *Controller* implementation that has two paths to execute b_{global} and b_{local} (see Fig. 8). Depending on the desired behavior either path or the combination is executed:

- *Global synchronization* is used to model waveforms, e.g. sinus sources or delayed controller outputs for feedback loops. A globally synchronized *Controller* begins a *computation path*, i.e. a *tick* triggers the execution of b_{global}, which is analogous to a *Sensor* implementation.
- *Local scheduling* is used for most feedback control algorithms, where the computation is triggered directly by the input values. Thus, no extra global clock is necessary. This enables the use of CPUs lacking a dedicated *synchronization* mechanism.
- *Mixed synchronization* with local scheduling and global synchronization is used for the optimization of *computation paths*. Therefore, the two paths are connected by a FIFO, which delays the signals until the next tick occurs. This is a common optimization principle for the synchronous design of digital hardware.

It is quite intuitive to implement two different mechanisms for synchronization and scheduling, and most synchronous approaches follow this concept [Benveniste et al. (2003)]. Taken together, *Controllers* (in cooperation with *Sensors*) are the *Virtual Path*'s building blocks for the deterministic implementation of distributed computation. The start of execution is always determined locally, by either the incoming signals or the local implementation of a *global clock*. With such a decoupled scheduling implementation, the distribution of controllers is simple and enables the flexible adaption of a net of controllers to a heterogeneous hardware architecture without a monolithic framework.

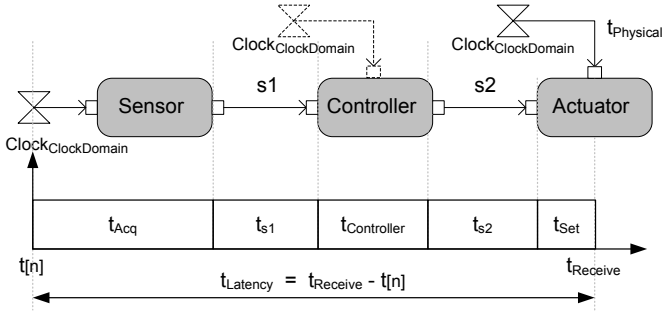

Fig. 9. Latency is the sum of all delays along the path.

5. ERROR DETECTION AND HANDLING

Proper synchronization is the first step towards determinism. However, errors such as imprecise measurements, propagated signal noise, and communication failures are not tackled by synchronization. Therefore,

- *systematic errors* and predictable *statistic* errors have to be eliminated by calibration or debugging, and
- *unpredictable statistic errors* have to be handled at runtime.

In the context of signal-oriented systems, statistic errors are known as signal noise or as lack of signal quality. The representation of signal quality is numerous, e.g. variance (i.e. RMS), amplitude deviation, binary confidence, or logarithmic measures such as entropy [Shannon (1948)].

The *Virtual Path* takes advantage of *domain knowledge* and provides the following simple but effective concept for error propagation and error handling:

- *Sensors* and *Controllers* rate the *quality* of all output signals and attach *quality* to each output signal. Thus, *quality* is propagated with the natural signal flow to the *Actuators*.
- *Actuators* check their input signals and validate *input values*, *quality*, and *timestamp*. In the case of an error, all *Actuators* coordinately turn to a safe state.

Thus, all errors are propagated to the actuators either explicitly by *quality* estimation or implicitly as late or missing events. The actuators handle errors by turning to a safe state, e.g. by setting the total energy impact to zero. Since only *Actuators* have an impact on the physical world, this concept guarantees a well-defined system state even in the presence of run-time errors.

5.1 Error Propagation And Detection

As mentioned before, the *Virtual Path* extends the event e (see Eqn. 1) by $q \in Q$, where Q is the set of *qualities*.

Due to the manifold representations of *quality*, the appearance of q is left open so that it can be defined by the application designers. In the current setup of the *MIRO Platform*, a simple binary confidence is implemented.

The quality, q, is more than the result of a real measurement. It expresses a component designer's estimation of the quality of an output signal, for which the designer takes responsibility. It manifests a contract between component designers, about whether to rely on a result or not.

Latency is defined as the accumulation of all communication and computation delays of a *computation path* from occurrence of a *tick* to the point of time when the *Actuator* sets the value (see Fig. 9). Thus, *latency* is the deviation at the end of the path between physical time and $t[n]$. Hence,

$$t_{Latency} = t_{Receive} - t[n], \tag{5}$$

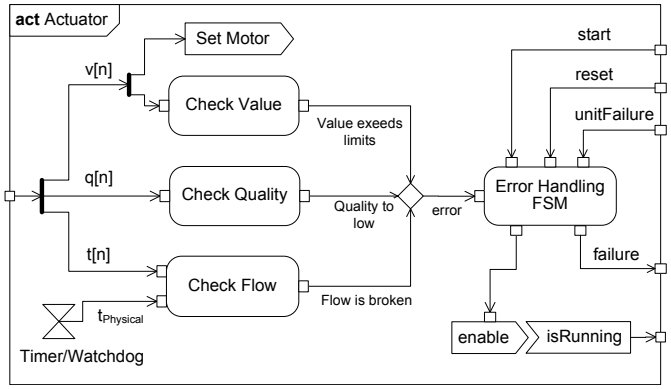

Fig. 10. An *Actuator* implements error detection and handling.

where $t_{Receive}$ is the current *physical time* measured by a continuous synchronized clock and $t[n]$ is defined by the *Clock Domain* of Eqn. 3.

Actuators observe their input signals to detect the following errors:

- input value error: e.g. $v_{min} < v[n] < v_{max}$
- quality error: $q[n] < q_{min}$
- total order violation: $t[n] \neq t[n-1] + T_s$
- latency error: $t_{Receive} - t[n] > Latency_{max}$

Actuators implement the detection and handling of those errors (see Fig. 10). Input value errors are detected by *Check Value* to inhibit operation outside of the actuator's specification. *Check Quality* observes the quality value. *Check Flow* detects *latency errors* and *total order violation*. Errors are handled by the *Error Handling FSM*, which represents the *Actuator's* operational state. This state determines whether the *Actuator's* motor is enabled (see below) and sets the signal *enable* accordingly. The sensor *isRunning* measures whether this request succeeds.

5.2 Global Error Handling

The mechanical structure and the safety constraints of an application define reasonable groups of conjoint actuators that form an *actuation unit*. An *actuation unit* has the following attributes:

- actuators must start and stop simultaneously
- all actuators stop on error and do not autonomously restart

Every actuator implements a finite state machine (FSM) that communicates with the other actuators via binary signals with a simple handshake protocol, so that the combination of actuators can be realized with basic logic operators. Fig. 13 depicts how merely two combinatorial operators implement the global state of an *actuation unit*. This global state is represented by the following two signals. The first, *unitRunning*, is the conjunction of the unit's actuators' *isRunning* signals. The second, *unitFailure*, is the disjunction of the unit's actuators' *failure* signals.

The *user* represents a command entity that issues requests to the *actuation unit*. The signal *start* requests to start/stop the unit's actuators. *reset* requests to reset any

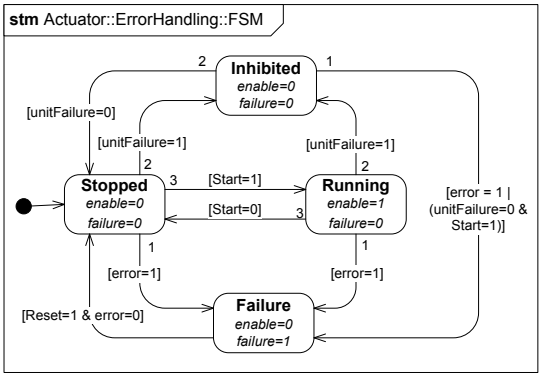

Fig. 11. The actuator's *Error Handling* is defined by a simple FSM.

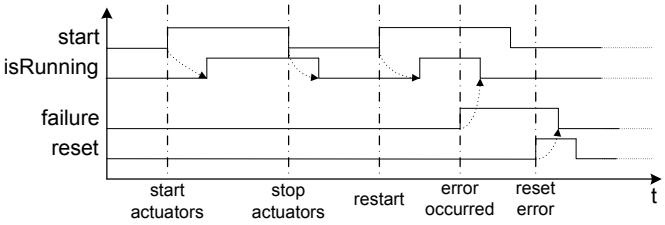

Fig. 12. The *Actuator* coordination protocol is asynchronous and not part of a *Clock Domain*.

stored errors. The *user* is notified about the global state of its *actuation unit*.

The actuator's FSM is driven by the user's request signals, the unit's *unitFailure*, and the result of the actuator's internal error validation, *error* (see Fig. 11). The four FSM's states are as follows:

- *Stopped:* The *Actuator* is disabled and ready to start.
- *Running:* The *Actuator* is enabled.
- *Failure:* The *Actuator* detected one of the following errors: input value error, quality error, total order violation, or latency error.
- *Inhibited:* The *Actuator* is disabled, since another actuator of the same unit detected an error.

The handshake protocol (see Fig. 12) for the coordination of the actuators' FSMs is asynchronous, i.e. its signals are

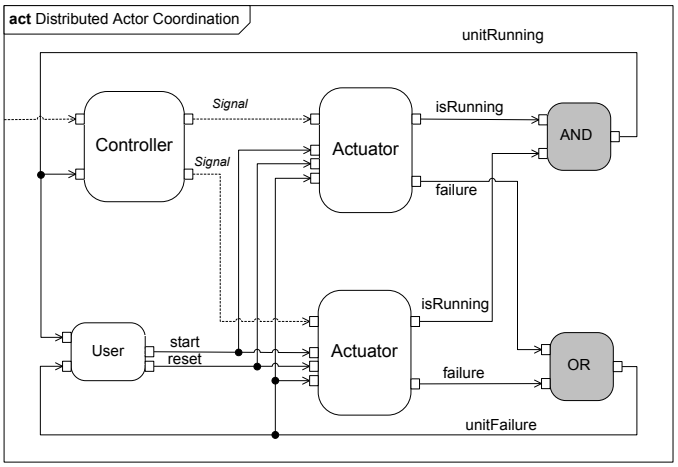

Fig. 13. The coordination of distributed *Actuators* is implemented with only two logical operators.

not part of a *Clock Domain*. Therefore, actuators of different *Clock Domains* can be combined to one actuation unit. Together, the simple handshake protocol and the *Error-Handling FSMs* provide the coordination of distributed actuators without requiring a complex robot controller framework.

6. CONCLUSIONS

Over the last five years, two generations of medical robotic systems have been successfully designed with the *Virtual Path* approach at DLR's Institute of Robotics and Mechatronics. The application of the domain model has enabled our designers to tackle the complexity of current mechatronic systems. The *Virtual Path*'s *model of interaction*, which defines synchronization, scheduling, and error handling, is an abstract infrastructure specification that constitutes a valid alternative to infrastructure design with a monolithic all-in-one framework.

The *Virtual Path* pragmatically "redefines" synchrony as "fast enough". Therefore, the path's overall latency is used as a measure for the deviation from the synchronous hypothesis. This balancing act of loosening the system specification for the sake of design flexibility is stabilized by the definition of explicit design roles. These roles specify the interaction of the basic system components so that an assembly thereof always constitutes a deterministic system.

The concept of passing all errors with the natural signal flow to the *Actuators* is very effective. This differs from many common approaches where monolithic robot control software is used. The *Virtual Path* does not intend to provide an elaborate safety concept. In contrast, it claims to be a minimal definition of determinism for heterogeneous systems. Therefore, higher level safety algorithms still have to be implemented on top of this foundation. These safety algorithms would, however, naturally benefit from the deterministic platform provided by the *Virtual Path*.

Beyond this, a designer that specifies the quality of a component's output signals immediately takes responsibility for that component's calculation results. This is an effective in-line specification mechanism, where constraints are defined within the source-code and appear at component interfaces. Hence, no additional paper specification, albeit useful, is required. This makes the *Virtual Path* a valuable building block for an agile design process for mechatronic systems.

To further evaluate the model, we are applying the *Virtual Path* method to the design of a novel integrated anthropomorphic hand-arm-system that will consist of more than 50 actuators [Grebenstein and van der Smagt (2008)]. Furthermore, we are working on the design of a semi-automatic mapping process that is able to generate a customized communication-infrastructure for a robotic system from an abstract specification. Safety certification is essential for medical devices such as the MIRO system and we plan to investigate how the agile design process of the *Virtual Path* can be applied to it.

REFERENCES

Agha, G.A. (1986). *Actors: A Model of Concurrent Computaton in Distributed Systems*. MIT Press, Cambridge, MA.

Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., and Sangiovanni-Vincentelli, A. (2003). Metropolis: An integrated electronic system design environment. *Computer*, 36(4), 45–52.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, 64– 83.

Benveniste, A. and Berry, G. (2001). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1270 – 1282.

Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003). Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 91(1).

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity In The Heart Of Software*. Addison-Wesley.

Franklin, G., Powell, J.D., and Workman, M.L. (1998). *Digital Control of Dynamic Systems*. Addison Wesley, 3rd edition.

Grebenstein, M. and van der Smagt, P. (2008). Antagonism for a highly anthropomorphic hand-arm system. *Advanced Robotics*, 22(1), 39–55.

Hagn, U., Nickl, M., Jörg, S., Passig, G., Bahls, T., Nothhelfer, A., Hacker, F., Le-Tien, L., Albu-Schäffer, A., Konietschke, R., Grebenstein, M., Warpup, R., Haslinger, R., Frommberger, M., and Hirzinger, G. (2008). The dlr miro: a versatile lightweight robot for surgical applications. *Industrial Robot: An International Journal*, 35(4), 324–336.

Hoare, C.A.R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8), 666–677. doi: http://doi.acm.org/10.1145/359576.359585.

Kienhuis, B., Deprettere, E.F., van der Wolf, P., and Vissers, K. (2001). A methodology to design programmable embedded systems - the y-chart approach. In *SAMOS: Systems, Architectures, Modeling, and Simulation*, volume 2268 of *LNCS*, 18–37. Springer Verlag.

Kopetz, H. and Bauer, G. (2001). The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 558–565.

Lee, E.A., Neuendorffer, S., and Wirthlin, M.J. (2003). Actor-oriented design of embedded hardware and software systems. In *Journal of Circuits, Systems, and Computers*, volume 12, 231–260.

Lee, E.A. and Sangiovanni-Vincentelli, A. (1998). A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 1217–1229.

Shannon, C.E. (1948). A mathematical theory of communication. *The Bell Systems Technical Journal*, 27, 379–423,623–656.